575 Software Engineering Design : Tutorial 2

Work in pairs.
In this tutorial you will write your first Java class hierarchy, including interfaces, abstract and concrete classes. You will explore some more Java Standard Library classes and work with recursive methods. You will also use your first Java Design Pattern, the Factory Pattern.

The class hierarchy that you will build aims to provide functionality to operate using arithmetic expressions over natural numbers (i.e., positive integers).

Note that your are provided a set of tests for you to use to check the correctness of your code. These tests will initially not compile as they refer to code that you will have to develop. Comment out all the tests and comment them back in as you progress through the different tasks described below.

Look at pages 2 onwards of this spec for details of how to get the skeleton code, test it, and submit.

Expression Subclasses

You are given code for the `Expression` interface and a `NaturalNumber` class that implements `Expression`. A `NaturalNumber` is an expression of depth 0 that consists of positive integers (i.e., positive whole numbers). Our first task is to implement two other sub-types of `Expression`: `Addition` and `Product`. These should be implemented as follows:

- Both classes should be in the same package as `NaturalNumber`
- Both classes should have a constructor that accepts two `Expression` parameters. In each case, the first argument is the left operand and the second is the right operand.
- Both classes should implement the `Expression` interface.
- Both classes should override the `toString()` method to pretty print the expressions.

Your code should allow for expressions that have various nested sub-expressions such as `(4+4)*(3+2)` and `4+(3*(2+3))`. The depth of these expressions are 2 and 3 respectively.

Factoring out Duplication

The two classes you have just written share various properties:

- They have similar structure in terms of fields.
- Their constructors are very similar.

Define an `abstract class BinaryExpression` that takes these common elements of `Addition` and `Product` classes. Define `BinaryExpression` as a subtype of `Expression`, so that all `BinaryExpression` objects are also of type `Expression`. Redefine `Addition` and `Product` classes to be subclasses of `BinaryExpression`.

Exceptions

Create two new `BinaryExpression` subclasses: `Subtraction` and `Division`. Ensure that both classes do not allow creating invalid natural number expressions. `Subtraction` should not allow building expressions that evaluated would return a non positive number (e.g., `3-5, 4-(2+2)`) and `Division` should not allow expressions that where the divisor does not exactly divide the dividend (e.g., `5/2`) or where the divisor is zero (e.g., `5/0`).
Create a subclass of `RuntimeException` called `InvalidNaturalNumber` class that has two further subclasses: `NonWholeNumber` and `NonPositiveNumber`. Use the former for invalid creation of division expressions and the later for invalid subtractions.

Modify class `NaturalNumber` to throw an appropriate exception.

Write a class `ExpressionFactory` with a static method `Expression randomExpression(int maxDepth)` that returns an expression of up to `maxDepth` randomly built from nested sub-expressions containing natural numbers of one digit. For instance `Expression.randomExpression(0)` will create expressions that are natural numbers (e.g, `5, 2, 9`, but not `12,` nor `65`). But `Expression.randomExpression(2)` may return expressions like `5`, `(2*4)/3`, or `6/3` but not `((1+1)*4)/3` which has depth 3. You can use `java.util.Random` and its method `nextInt()` to support random number generation.

Your function `randomExpression` should catch all `InvalidNaturalNumber` and resolve them in some way you find suitable.

Sorting Expressions

You have seen how to define and implement your own interface type in Java. The Java API also contains "off the shelf" interfaces that your classes can implement. The Java API also provides classes with methods that rely on having parameters that implement these interfaces. For instance, the static method `Arrays.sort(...)` sorts an array that contains elements that implement the `java.lang.Comparable` interface. You can read about both the sorting method and the interface from the online Java API documentation.

In a nutshell, objects of any class implementing `Comparable` can be sorted using `Arrays.sort(...)`.

Your next task is to implement `Comparable` for all `Expression` types, so that they are sortable. Firstly, make `Expression` a subtype of `Comparable`. Interfaces can inherit from one another like classes, using the same notation. So, change the declaration of Expression to be:

```
public interface Expression extends Comparable<Expression>...
```

The `<Expression>` is necessary because `Comparable` is generic. If we have not talked about it in the lectures, don't worry about this now. For now you need to decide how you are going to implement `Comparable<Expression>`. The interface has a single method:

```
public int compareTo(Expression other)
```

which must be defined for all Expressions. Decide in which class or classes this method should be defined. A call `this.compareTo(that)` returns:

- a positive int if `this` is "greater than" `that`
- a negative int if `this` is "less than" `that`
- zero if `this` is "equal to" `that`

You should use the value of the expression (the integer value obtained by evaluating the expression) to determine the order. As an example, calling `compareTo` on an `Expression (3*4)` using as a parameter an `Expression` that encodes `(1+1)*1` will return a positive `int`.

Create a class `ExpressionArrayFactory` that provides a method `randomExpressionArray(int maxSize, int maxDepth)` that returns an array of expressions of random of size `maxSize` populated with random expressions of depth up to `maxDepth`.

Note that, according to the documentation of the Comparable interface, `compareTo(Expression other)` must be consistent with `equals(Object o)`. This is something that the compiler does not check.

Override `equals(Object o)` so that for any expression `e` and any object o, `e.equals(o)` implies that the actual type of `o` is a subtype of `Expression` and `e.compareTo((Expression) o) == 0`. Note that `((Expression) o)` is down-casting the reference `o` to an object of type `Object`, to a reference of type `Expression`.

Questions

Consider the following questions. For each one, we strongly suggest first thinking what the answer might be, and then modifying your code to see if you were right or wrong.

- Why did we choose to make `BinaryExpression` an abstract class? What changes to the code are needed if we make it concrete? Are they sensible?

- Why did we we chose to create exception classes that are subclasses of `RuntimeException` rather than `Exception`? What changes to the code would need to be made if we decide to make them subclasses of `Exception`?

Getting The Skeleton Code

Get the outline from GitLab:

```
git clone https://gitlab.doc.ic.ac.uk/lab1819_spring/575_exercise2_LOGIN.git
```

You should work in a pair, but only one person needs to clone the repository. The best way to work is two people sitting at one computer. Work together on the code and make commits back to this repository. All the commits will be in the name of whoever is logged in, but that is ok. A good convention is to add both your names (or initials) to each commit comment.

Project Structure

There are two source folders, **src** and **test**. Inside each of these folders you already have some classes which we are aiming to develop more fully.

Importing the code into an IDE…

For **IntelliJ IDEA**: you can *Import Project*, browse to cloned directory, *Create project from existing sources,* then keep clicking *Next* until finished. If no JDK is set, choose a >= 1.7 JDK.

Build Results

The build has a number of stages and checks a number of qualities of your code. All of these have to pass for the whole build to pass.

Compilation: compiles your code - obviously your code has to compile correctly.

Test Results: runs all of our tests - if any of them fail, it fails the build.

Checkstyle: checks a number of things about the style of your code, following standard conventions for naming, layout etc.

Submission

When you have finished, make sure you have pushed all your changes to GitLab (source code only, not generated files under *target*). You can use LabTS (https://teaching.doc.ic.ac.uk/labts) to test the version of your work that is on GitLab. Then you need to submit a revision id to CATe so that we know which version of your code you consider to be your final submission.

Autotest with LabTS

The LabTS build has a number of stages and checks a number of qualities of your code:

Compilation: compiles your code - does it compile correctly?

Test Results: runs all of your test. We are not providing additional tests this time.

Style: we run Checkstyle to check a number of things about the style of your code, following the **Google Java Style** standard conventions for naming, layout etc.

Deadline

You are strongly encouraged to complete the exercise during the lab session on Thursday afternoon. However it is possible to submit up until **9am (in the morning!) Monday 28th October** (think of this as Sunday evening).

Assessment

The first 50% of the marks are for the automated tests and checks:

- Code compiles
- Tests pass
- Style checks pass

If you pass these automated checks then you are eligible for the other 50% of the marks, based on:

- Code meets requirements